Chapter 7
Lexicase Selection Beyond Genetic Programming

Blossom Metevier, Anil Kumar Saini, and Lee Spector

7.1 Introduction

Lexicase selection is a selection algorithm for evolutionary computation systems, used to determine which individuals will be permitted to contribute to future generations in the evolutionary process. Although it has been used for survivor selection [8], its primary use has been as a *parent* selection algorithm, selecting individuals to be provided as inputs to genetic operators. The genetic operators, such as mutation and crossover, use the selected parents as source material out of which to construct children.

Lexicase selection selects individuals by filtering a pool of individuals which, before filtering, typically contains the entire population. The filtering is accomplished in steps, each of which filters according to performance on single test case (input/output pair). The test cases are considered one at a time in random order. Lexicase selection has been tested most extensively in genetic programming systems, where it has been shown to outperform other selection methods in several contexts [2–5, 7, 9–11]. However, the effectiveness of lexicase selection in other settings has not been fully explored.

In this paper, we investigate the utility of lexicase selection in traditional genetic algorithms with linear, fixed-length genomes. We chose this framework in part

B. Metevier · A. K. Saini
College of Information and Computer Sciences, University of Massachusetts, Amherst, MA, USA
e-mail: bmetevier@umass.edu; aks@cs.umass.edu

L. Spector (✉)
School of Cognitive Science, Hampshire College, Amherst, MA, USA

College of Information and Computer Sciences, University of Massachusetts,, Amherst, MA, USA
e-mail: lspector@hampshire.edu

W. Banzhaf et al. (eds.), *Genetic Programming Theory and Practice XVI*,
Genetic and Evolutionary Computation,
https://doi.org/10.1007/978-3-030-04735-1_7

because the large literature of traditional genetic algorithms provides context for the interpretation of our results that is both broad and deep.

The problems to which we apply traditional genetic algorithms, using several parent selection methods, are randomly generated Boolean constraint satisfaction problems [1]. Although the problems are derived from Boolean satisfiability problems, the problem-solvers (in this case, the genetic algorithms) are not given access to the constraints themselves, or to the variables that they contain. Rather, the problem-solvers are given access only to a procedure that determines whether each constraint is satisfied by an assignment of truth values to all variables. Crucially, the problem-solvers are given no information about which constraints may depend on which others. This design is intended to allow the problems serve as abstractions of problems in many real-world domains, in which we can tell whether or not a candidate solution satisfies a constraint, but we have no further information about the nature of the constraint, or of the ways in which different constraints might be interdependent.

In this chapter, we present the results of experiments using the traditional genetic algorithm, with lexicase selection, to solve Boolean constraint satisfaction problems. We compare the performance of the algorithm using lexicase selection to the performance of the same algorithm run with more traditional selection algorithms, specifically fitness-proportionate selection and tournament selection (with several tournament sizes).

In the following sections we first describe the lexicase selection algorithm that is the focus of this investigation. We then describe the Boolean constraint satisfaction problems that we use for our experiments, and our experimental methods. We then present our results, and discuss their implications for future work.

7.2 Lexicase Selection

Lexicase selection is a method by which individuals can be selected from a population for use as the source material out of which genetic operators, such as mutation and crossover, construct offspring for the following generation. Lexicase selection is distinctive in that it allows selection to depend on multiple assessment criteria and all of their combinations, without requiring that these criteria be aggregated into overall "fitness" values. This is different from the selection methods used traditionally in genetic programming, which require the assignment of a single scalar value to each candidate solution in order to guide search.

In most of the prior work on lexicase selection, it has been used in genetic programming systems to select parent programs that are then subjected to variation to produce the next generation of programs. In this context, the assessment criteria are generally the errors of the program on different inputs, which are often referred to, in the genetic programming literature, as "fitness cases" or "test cases."

Algorithm 1: Lexicase selection

```
Result: Individual to be used as a parent
candidates := the entire population
cases := list of all test cases in a random order
while True do
    candidates := candidates performing best on the first case
    if only one candidate exists in candidates then
        return that candidate
    end
    if cases is empty then
        return a randomly selected candidate from candidates
    end
    delete the first case from cases
end
```

In lexicase selection, each time a parent is needed, a pool of individuals, which initially contains the entire population,[1] is winnowed in successive stages until a single individual remains and is selected. In the first stage, only the individuals that perform best over a randomly chosen test case are retained. If more than one individual remains, a second randomly chosen test case is used for the next stage of winnowing. This process repeats until only a single individual remains, or until the test cases have been exhausted, in which case a random individual from the remaining pool is selected. Pseudocode for the most commonly used form of lexicase selection is provided in Algorithm 1.

Sometimes, lexicase selection chooses individuals with performance that is good over only a small number of test cases. Many of these "specialists" would, under many selection methods that require aggregation of performance on all test cases into single scalar values, rarely be selected for reproduction and variation. This would often be the case even if one of the cases solved by the specialist was difficult for a majority of the population to solve. The reason for this is the assumption of uniformly distributed selection pressure, with all parts of a problem being equally hard."Specialists" are ignored even though they are better at certain subsets of the problem and may contain a partial solution to the task at hand. By allowing these "specialists" to contribute to the next generation, lexicase selection allows for offspring that may contain solutions to a particular subset of the problem. Comparisons of lexicase selection to other methods developed with similar motivations, such as "implicit fitness sharing" and "deterministic crowding," are presented elsewhere [5, 7].

Several variants of the lexicase selection method have also been developed and studied. For example, *epsilon lexicase selection*, a variant in which candidates that are not strictly "best" on the current case but which are "close enough" (within

[1] It is also possible to limit the initial pools in various ways. When the initial pool contains the entire population, which is the best-studied setting, we refer to the algorithm more specifically as "global pool" lexicase selection.

epsilon, for some definition of epsilon) has been developed and shown to be particularly effective on problems with floating-point errors. Other variants have been developed and explored in previous instances of the *Genetic Programming Theory and Practice* workshop [12]. For the present study, however, we used the simplest and most standard version of the method, as described above.

7.3 Problems

7.3.1 Boolean Constraint Satisfaction

The problems used for the experiments in this study are Boolean constraint satisfaction problems [1] that are randomly generated based on three parameters: a total number of variables v, a number of constraints c, and a number of clauses per constraint n.

Each clause is a disjunction of three literals, each of which is either a variable or a negated variable. Each constraint is a conjunction of clauses, and a problem is a conjunction of constraints. An assignment of truth values to the variables is a solution if all of the constraints evaluate to true in the context of the assignment.

These problems are similar in some respects to Boolean satisfiability problems expressed in 3-CNF (conjunctive normal form, with three literals per clause), with the clauses grouped to form constraints. However, unlike the case with standard satisfiability problems, such as those used in SAT-solver competitions [6], we do not allow our problem solvers to see the formulae themselves, or to have any information about which variables appear in which constraints. The problem-solvers can evaluate an assignment of truth values to the variables with respect to each constraint, determining whether or not each constraint is satisfied by the assignment, but this is the only information that the problem solver receives about the problem.

7.3.2 Random Problem Generation

We generate a problem by starting with a random assignment of truth values to all variables. This assignment will be a solution to the generated problem, but we will discard it after generating the problem, and it will be the task of the problem solver to re-discover the assignment, or to discover another assignment that also satisfies all of the constraints in the problem.

Once we have a random assignment of truth values to all variables, we generate the problem itself with a simple, iterated generate-and-test algorithm: We create a random set of constraints of the specified size (which may include duplicate clauses, possibly in different constraints), and we check to see if it evaluates to true with

Table 7.1 Problem parameters

Parameter	Value
Number of variables (v)	20, 30, 40
Number of constraints (c)	8, 12, 16, 32
Number of clauses per constraint (n)	20, 25, 30, 35, 40
Number of problems per combination of v, c, and n	15
Number of runs per method per problem	50
Total number of runs per method per combination of v, c, and n	750

Table 7.2 Genetic algorithm parameters

Parameter	Value
Population size	200
Number of generations	500
Mutation operator	Bit-flip
Probability of mutation	0.1
Crossover operator	One-point
Probability of crossover	0.9

the given assignment. If it does, then we use the constraints as a problem for our experiments; if it doesn't, then we randomly generate a new set of constraints, repeating the process until we find one that is satisfied by the assignment.

For each (v, c, n) triple, we generated 15 different problems. Each problem-solving method was run 50 times on each of these problems, resulting in 750 runs per problem-solving method for each combination of v, c, and n. Results were evaluated by averaging over all 750 runs for each parameter combination. The specific parameters used for generating problems, and for the numbers of runs conducted on each problem with each method, are shown in Table 7.1.

7.4 Experimental Methods

7.4.1 Genetic Algorithm

Our problem-solving methods were all instances of the same genetic algorithm, with identical parameters (shown in Table 7.2) except for the parent selection method.

Individuals in the population were truth assignments, with genomes consisting of genes for each variable, indicating whether that variable had a value of true or false in the specified assignment.

We used a generational genetic algorithm that began with a population of random individuals, and then entered a cycle in which, for each generation, all individuals were tested for errors, parents were selected for the production of children on the basis of those errors, and children were produced by varying the selected parents.

7.4.2 Variation

At the variation step, individuals selected to serve as parents were first (possibly) subjected to crossover and then (possibly) to mutation.

The standard one-point method was used as the crossover operator, allowing parent recombination at a randomly chosen crossover point. The crossover rate was 0.9, meaning that 90% of children were produced by crossover.

For mutation, a bit-flip mutation operator was used, allowing for a randomly chosen bit in a chromosome to be flipped. The mutation rate was 0.1, meaning that 10% of children were subject to mutation.

7.4.3 Parent Selection

We compared lexicase parent selection, tournament parent selection, and fitness proportionate parent selection. All selection methods performed selection with replacement; that is, the same individual might be selected to be a parent several times in the same generation.

For tournament selection and fitness proportionate selection, an individual's total error value was determined from the number of constraints it satisfied. If an individual satisfied all constraints, then its error was 0. Otherwise, its error value was the number of constraints that it did not satisfy.

For tournament selection with an integer-valued tournament size t, we first form a tournament set of t individuals, each of which is chosen with uniform probability (with possible duplication) from the entire population. We then return, as the selected parent, the individual in the tournament set that has the lowest total error.

Higher tournament sizes make tournament selection more selective, in the sense that individuals with high total error are less likely to be selected, while lower tournament sizes make it less selective. Because our preliminary experiments showed that less selective settings appeared to perform better, we wanted to consider methods even less selective than tournament selection with tournament size 2, which is normally considered to be the minimum, since with tournament size 1 tournament selection is equivalent to selecting individuals entirely randomly. For this purpose we adopted the convention that for a non-integer-valued tournament size t between 1 and 2 we would use tournament size 2 with probability $t - 1$, and select a parent entirely randomly otherwise. For example, with $t = 1.25$, 25% of the time we will choose 2 individuals randomly and return, as the selected parent, the one with the lower total error; the remaining 75% of the time we will return, as the selected parent, a parent chosen with uniform probability from the entire population.

We performed fitness-proportionate selection in the standard way: The probability of selection for an individual i that satisfies s_i constraints is s_i divided by

sum of s_j for all individuals j across the population. In the degenerate case of no individuals satisfying any constraints, which would produce a denominator of zero, an individual is selected at random.

7.5 Results

7.5.1 Success Rates by Parent Selection Method

Our primary results are shown in Table 7.3, comparing success rates of the genetic algorithm when run with fitness proportionate parent selection, tournament parent selection (with tournament size 2), and lexicase parent selection.

Table 7.3 contains a row for every combination of number of variables (v) and number of constraints (c). All runs with a specified value of v and c are aggregated in the corresponding line, regardless of the number of clauses per constraint (n). Because we conducted 750 runs with each combination of v, c, and n for each parent selection method, and because we conducted experiments with 5 different values of n (see Table 7.1), each row in Table 7.3 reports data from $5 * 750 = 3750$ runs with each of the three parent selection methods listed in the table.

The numbers reported in Table 7.3 are success rates, defined as the proportion of the total runs that produced a successful solution (with error vector consisting only of zeros). Lexicase selection produces the highest success rate in every case, and the improvement provided by lexicase selection is statistically significant in most cases.

Table 7.3 Success rate for the genetic algorithm with fitness proportionate, tournament (size 2), and lexicase parent selection for each studied combination of v (number of variables) and c (number of constraints)

Number of variables (v)	Number of constraints (c)	Fitness proportionate	Tournament (size 2)	Lexicase
20	8	0.835	0.867	0.992
20	12	0.940	0.954	1.000
20	16	0.980	0.987	1.000
20	32	0.999	1.000	1.000
30	8	0.415	0.475	0.889
30	12	0.614	0.697	0.995
30	16	0.815	0.869	1.000
30	32	0.983	0.995	1.000
40	8	0.205	0.257	0.689
40	12	0.224	0.310	0.927
40	16	0.433	0.576	0.993
40	32	0.861	0.944	1.000

Underlines indicate statistically significant improvements, determined using a pairwise chi-square test with Holm correction and $p < 0.05$

The only cases in which the improvement is not significant are those in which all the selection algorithms approach a perfect success rate.

7.5.2 Success Rates by Tournament Size

Because the success rate of tournament selection is better or equal to fitness proportionate selection, many of our analyses in the remainder of this paper will compare lexicase selection only against tournament selection. Furthermore, because tournament selection is itself parameterized by the tournament size, we conducted additional experiments to compare performance across settings with different tournament sizes.

Table 7.4 shows the results of these runs. We again conducted 3750 runs for each combination of number of variables (v) and number of constraints (c), across the range of values for number of clauses per constraint (n) given in Table 7.1. We conducted runs for tournament sizes ranging from 1.25 to 8, with non-integer-valued tournament sizes handled as described in Sect. 7.4. The runs with tournament size 2 were independent of those conducted for the experiments documented in Table 7.3, so the numbers differ between the two tables, but not by much.

From Table 7.4 it appears that the most effective tournament size is around 1.5 or 2, and that larger tournament sizes perform poorly. None of the tested tournament sizes performs better than lexicase selection.

Table 7.4 Success rate for different tournament sizes

Number of variables (v)	Number of constraints (c)	Tournament size 1.25	Tournament size 1.5	Tournament size 2	Tournament size 4	Tournament size 8
20	8	0.850	**0.860**	0.856	0.818	0.777
20	12	0.948	0.955	**0.959**	0.952	0.934
20	16	0.982	0.987	0.988	**0.989**	0.979
20	32	**1.000**	**1.000**	0.999	**1.000**	0.999
30	8	0.443	**0.485**	0.471	0.428	0.367
30	12	0.644	0.702	**0.773**	0.712	0.618
30	16	0.850	**0.888**	0.879	0.846	0.766
30	32	0.993	**0.996**	**0.996**	0.990	0.974
40	8	0.226	**0.271**	0.137	0.120	0.105
40	12	0.254	**0.322**	0.293	0.245	0.213
40	16	0.510	**0.614**	0.503	0.423	0.335
40	32	0.938	**0.958**	0.901	0.794	0.680

Boldfaced numbers indicate the highest success rate in a particular row

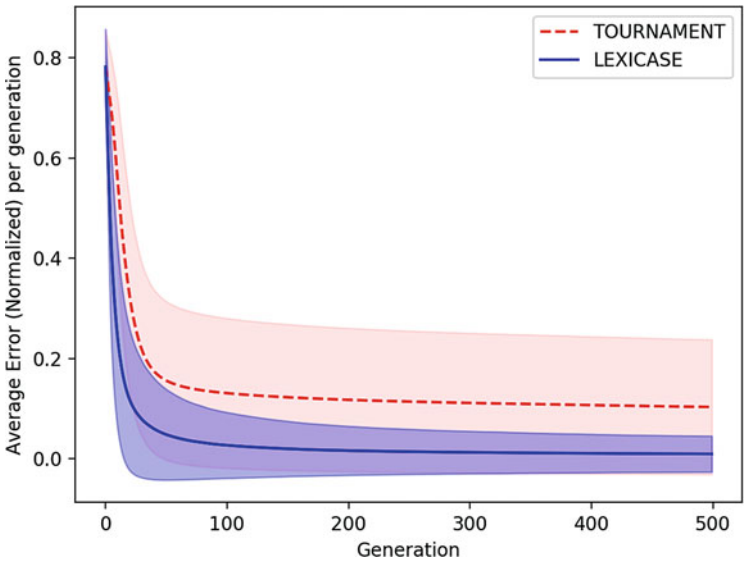

Fig. 7.1 Average error per generation for runs with lexicase selection and tournament selection with tournament size 2, with 50% confidence intervals

7.5.3 Errors over Evolutionary Time

Other features of the data produced by the runs described above, in Sect. 7.5.1, may be revealing, aside from the success rates described above.

In Fig. 7.1 we show the normalized average error across all runs. Here each error has been normalized by the total number of constraints (maximum error) used for that particular run. For a particular generation, the error has been averaged across the runs which were active up to that generation.

Here we see that lexicase selection not only produces lower errors, but also that lower errors are reached earlier in evolutionary time. We also see that both methods make most of their gains quite early in evolutionary time, with few improvements occurring after 100 generations.

7.5.4 Mean Least Error

Figure 7.2 shows the mean lest error (MLE) for each combination of number of variables (v) and number of constraints (c). MLE is defined as the average of the error values of the lowest-error individuals in each of the runs:

$$MLE = (1/N) \sum_i error(best\_ind_i),$$

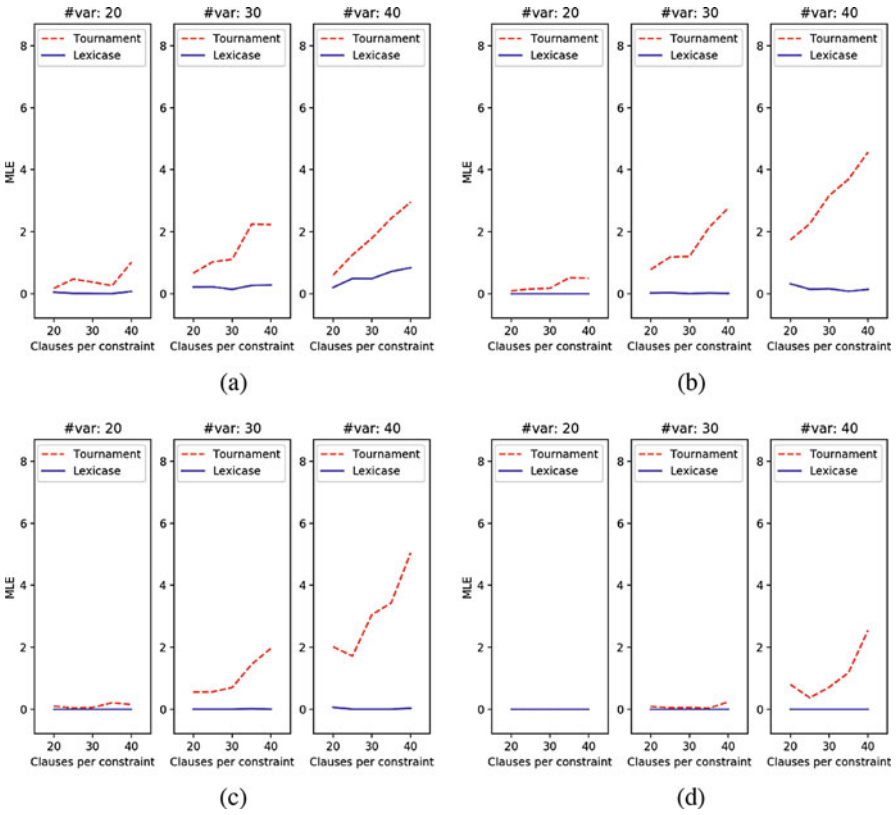

Fig. 7.2 Mean Least Error (MLE) for each combination of parameters. The number C denotes the number of constraints used for the corresponding plot. (**a**) $C = 8$, (**b**) $C = 12$, (**c**) $C = 16$, (**d**) $C = 32$

where $best\_ind_i$ is the individual having lowest total error in a given run i.

These plots show that tournament selection not only fails to solve problems in many cases, but also that the best errors achieved in the failing runs are often quite high. This effect is particularly pronounced for runs with large numbers of clauses per constraint.

7.5.5 Success Generations

Figure 7.3 shows the success generation—that is, the generation in which the genetic algorithm was able to find a zero-error solution—for each combination of number of variables (v) and number of constraints (c), averaged over the runs in which the genetic algorithm was able to find a solution.

Here we see that even when the genetic algorithm with tournament selection was able to find a solution, it generally required more generations to do so than did the genetic algorithm with lexicase selection.

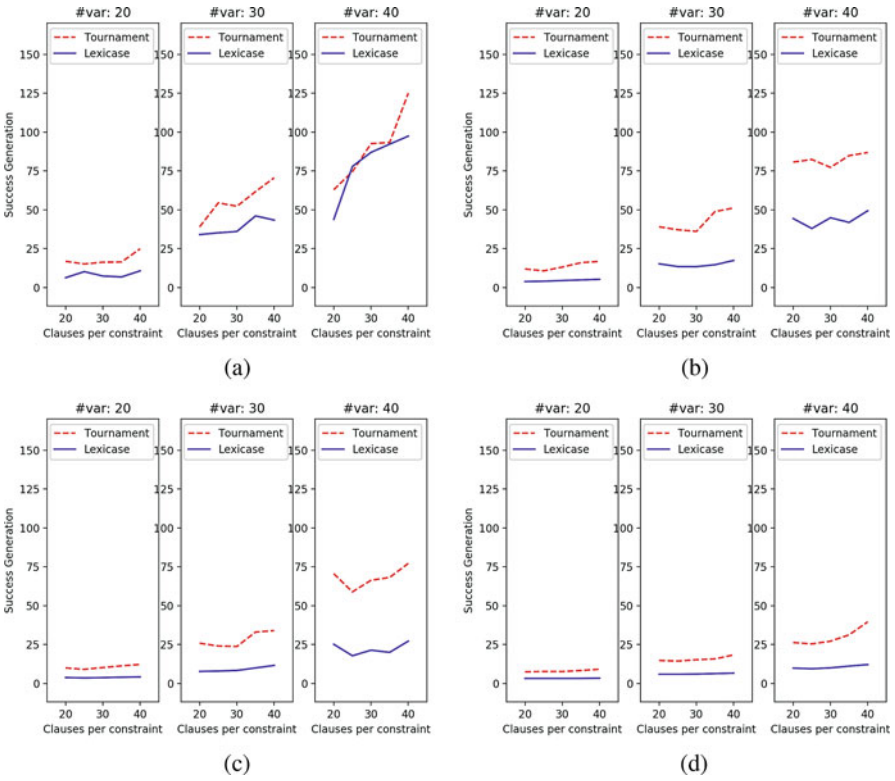

Fig. 7.3 Success Generation for each combination of parameters. The number C denotes the number of constraints used for the corresponding plot. (**a**) $C = 8$, (**b**) $C = 12$, (**c**) $C = 16$, (**d**) $C = 32$

7.5.6 Diversity over Evolutionary Time

Figure 7.4 shows the average number of unique chromosomes (individuals) in the population over evolutionary time, aggregated over all parameter combinations, and grouped by selection method and whether each method found a solution to the problem or not. We see from these plots that, with the exception of brief periods at the starts of runs, lexicase selection maintains relatively high diversity throughout the process of evolution, both in successful and in unsuccessful runs.

7.6 Discussion

The results presented above show that, for the Boolean constraint satisfaction problems studied here, the traditional genetic algorithm performs better with lexicase

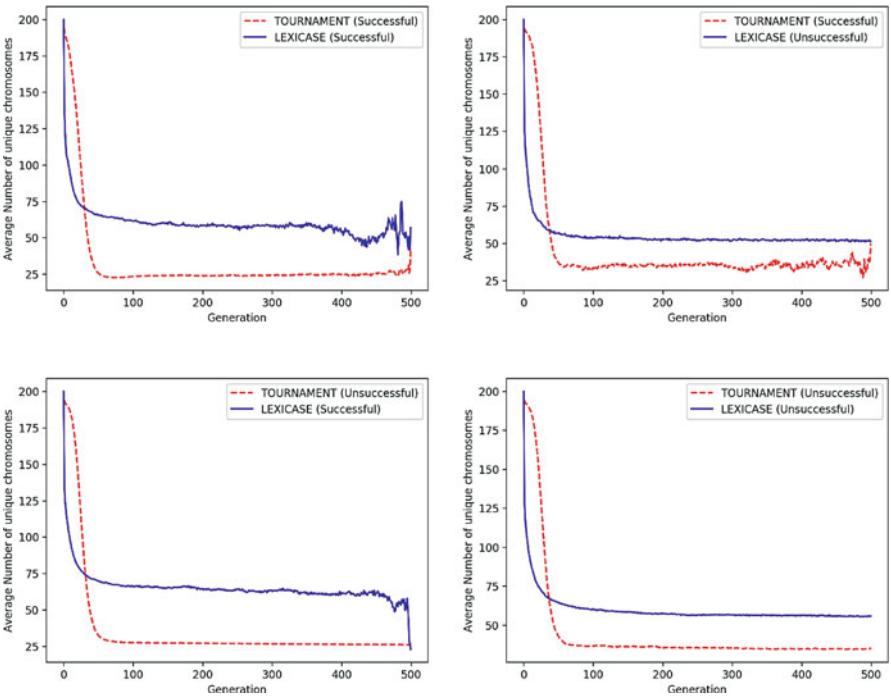

Fig. 7.4 Average number of unique chromosomes (individuals) in the population, over evolutionary time, under different conditions

parent selection than with tournament parent selection or fitness proportionate parent selection. In these experiments, lexicase selection found solutions more frequently and in fewer generations than did the other parent selection methods.

In addition, more diverse populations were maintained under lexicase selection, although the results here do not say anything definitive about the causal relations that may hold between diversity, success rate, and the number of generations required to find solutions.

With respect to the central question of this study, about whether lexicase selection has utility outside of genetic programming, these results suggest a positive answer: Lexicase selection does appear likely to have broader applicability than has been demonstrated previously.

The problems studied in this investigation were artificial, but they were designed to have features that resemble those many real-world problems. More specifically, the problems studied here were designed to be resemble real-world problems in which the goal is to satisfy many constraints simultaneously, but in which both the constraints themselves, and their interdependencies, are opaque.

For the problems studied here, the problem-solver is given access to a procedure that indicates whether or not each constraint is satisfied by a candidate solution, but it has no other information about the nature of the constraints or about shared

components or structure among multiple constraints. To the extent that a real-world problem fits this characterization, the results here suggest that lexicase parent selection could help to solve it.

Nonetheless, lexicase parent selection is probably not appropriate for all problems. For example, it seems unlikely that it would work well on problems that involve only a single constraint. In these cases, only the single individual in the population with the best performance on that constraint (or other individuals with the same performance) could be selected to serve as a parent. It seems reasonable to assume that this would undermine population diversity, making it more difficult to find solutions. Problems with more than a single constraint, but not many more, may be a poor match to lexicase parent selection for the same reason.

Previous work has also shown that lexicase parent selection sometimes performs poorly on problems with floating-point error values. The epsilon lexicase selection method appears to address this problem quite well [9], and it seems reasonable to assume that it would also work well on floating-point versions of the constraint satisfaction problems presented here.

One exciting avenue for future work would be an investigation of whether lexicase selection may have even broader applicability, perhaps extending beyond evolutionary computation altogether. Other machine learning methods might also be able to take advantage of the core idea of lexicase selection, that whenever we must make a decision based on the quality of candidate solutions, instead of aggregating multiple measures of quality into single, scalar values, we may instead consider them one at a time, in random order.

The ways in which this core idea of lexicase selection can be fleshed out will differ from one machine learning method to another, and we cannot yet provide definitive guidance on how it should be done for any specific methods outside of evolutionary computation. The results presented here, however, lead us to speculate that some such efforts will be rewarded with improvements in the problem-solving capabilities of the machine learning methods to which they are applied.

Acknowledgements We thank other members of the Hampshire College Institute for Computational Intelligence, along with other participants in the Genetic Programming Theory and Practice workshop, for helpful feedback and stimulating discussions.

This material is based upon work supported by the National Science Foundation under Grant No. 1617087. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

1. Craenen, B.G.W., Eiben, A.E., van Hemert, J.I.: Comparing evolutionary algorithms on binary constraint satisfaction problems. IEEE Transactions on Evolutionary Computation **7**(5), 424–444 (2003). https://doi.org/10.1109/TEVC.2003.816584

2. Helmuth, T., McPhee, N.F., Spector, L.: Effects of Lexicase and Tournament Selection on Diversity Recovery and Maintenance. In: Companion Proceedings of the 2016 Conference on Genetic and Evolutionary Computation, pp. 983–990. ACM (2016). http://dl.acm.org/citation.cfm?id=2931657
3. Helmuth, T., McPhee, N.F., Spector, L.: The impact of hyperselection on lexicase selection. In: Proceedings of the 2016 Conference on Genetic and Evolutionary Computation, pp. 717–724. ACM (2016). http://dl.acm.org/citation.cfm?id=2908851
4. Helmuth, T., McPhee, N.F., Spector, L.: Lexicase selection for program synthesis: a diversity analysis. In: Genetic Programming Theory and Practice XIII, pp. 151–167. Springer (2016)
5. Helmuth, T., Spector, L., Matheson, J.: Solving Uncompromising Problems with Lexicase Selection. IEEE Transactions on Evolutionary Computation **19**, 630–643 (2014). https://doi.org/10.1109/TEVC.2014.2362729
6. Järvisalo, M., Berre, D.L., Roussel, O., Simon, L.: The international sat solver competitions. AI Magazine **33**, 89–92 (2012)
7. La Cava, W., Helmuth, T., Spector, L., Moore, J.H.: A probabilistic and multi-objective analysis of lexicase selection and -lexicase selection. Evolutionary Computation (2018). https://doi.org/10.1162/evco_a_00224. In press.
8. La Cava, W., Moore, J.: A general feature engineering wrapper for machine learning using epsilon-lexicase survival. In: M. Castelli, J. McDermott, L. Sekanina (eds.) EuroGP 2017: Proceedings of the 20th European Conference on Genetic Programming, *LNCS*, vol. 10196, pp. 80–95. Springer Verlag, Amsterdam (2017). https://doi.org/10.1007/978-3-319-55696-3_6
9. La Cava, W., Spector, L., Danai, K.: Epsilon-Lexicase Selection for Regression. In: Proceedings of the Genetic and Evolutionary Computation Conference 2016, GECCO '16, pp. 741–748. ACM, New York, NY, USA (2016). http://doi.acm.org/10.1145/2908812.2908898
10. McPhee, N.F., Casale, M.M., Finzel, M., Helmuth, T., Spector, L.: Visualizing Genetic Programming Ancestries. In: Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion, pp. 1419–1426. ACM (2016). http://dl.acm.org/citation.cfm?id=2931741
11. Spector, L.: Assessment of problem modality by differential performance of lexicase selection in genetic programming: a preliminary report. In: Proceedings of the 14th annual conference companion on Genetic and evolutionary computation, pp. 401–408. ACM (2012)
12. Spector, L., La Cava, W., Shanabrook, S., Helmuth, T., Pantridge, E.: Relaxations of lexicase parent selection. In: W. Banzhaf, R.S. Olson, W. Tozier, R. Riolo (eds.) Genetic Programming Theory and Practice XV, pp. 105–120. Springer International Publishing, Cham (2018)